Local First Architecture - Detail

Local-First Hierarchical Sync Platform Architecture

Executive Summary

This document describes the architecture for a **local-first synchronization platform** that enables Progressive Web Applications (PWAs) to work offline while seamlessly syncing data across devices, teams, and organizations when connectivity is available.

The platform addresses a fundamental challenge in modern application development: **users expect apps to work everywhere, all the time, regardless of network conditions**. Traditional cloud-first architectures fail when networks are slow, unreliable, or unavailable. Our local-first approach inverts this model—apps work offline by default, and sync becomes an enhancement rather than a requirement.

Key Value Proposition:

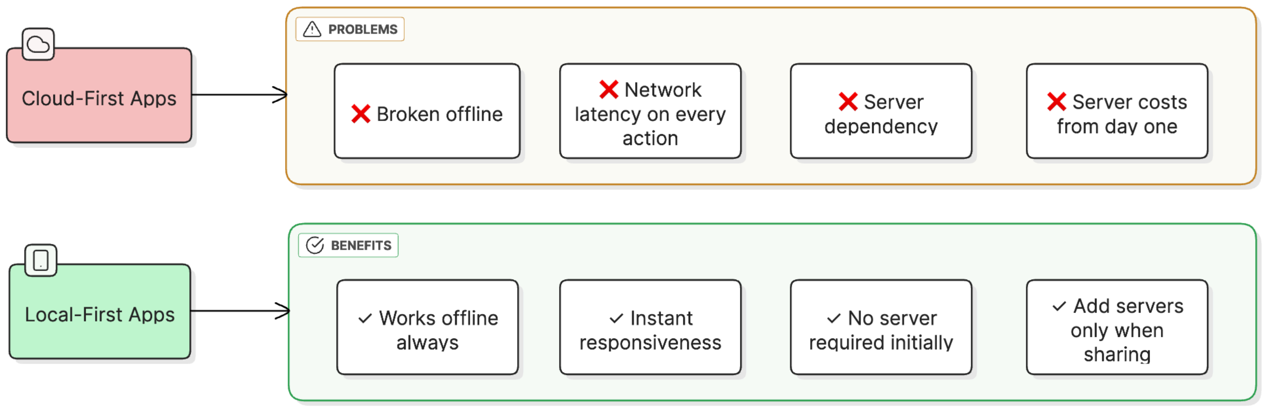
Stakeholder	Value Delivered
End Users	Apps that never show spinners, work on planes/subways, and don't lose data
Developers	Simple SDK that handles sync complexity; focus on features, not infrastructure
Organizations	Data sovereignty, privacy controls, and progressive scaling from free to enterprise
Platform Operators	Proven technology stack (PouchDB/CouchDB) with minimal custom code to maintain

---

1. The Problem

1.1 Why Local-First Matters

Modern web and mobile applications face a fundamental tension:



Real-world scenarios where cloud-first fails:

Scenario	Cloud-First Experience	Local-First Experience
Airplane/subway	App unusable	Full functionality
Poor connectivity (rural, developing markets)	Slow, frustrating	Instant response
Server outage	Complete downtime	Users unaffected
Collaborative editing	Conflicts lost or overwritten	Automatic merging
Privacy-sensitive data	Must trust cloud provider	Data stays local until explicitly shared

1.2 The Sync Problem

Building sync is notoriously difficult. Applications that attempt custom sync solutions typically face:

- **Conflict resolution**: What happens when two users edit the same document offline?
- **Partial connectivity**: How do you handle intermittent networks?
- **Data consistency**: How do you ensure all devices eventually converge?
- **Privacy boundaries**: How do you control what data syncs where?
- **Scale**: How do you go from personal use to enterprise?

Most teams either avoid offline support entirely, or spend months building fragile custom sync that breaks in edge cases.

1.3 Our Solution

We provide a **sync platform** that:

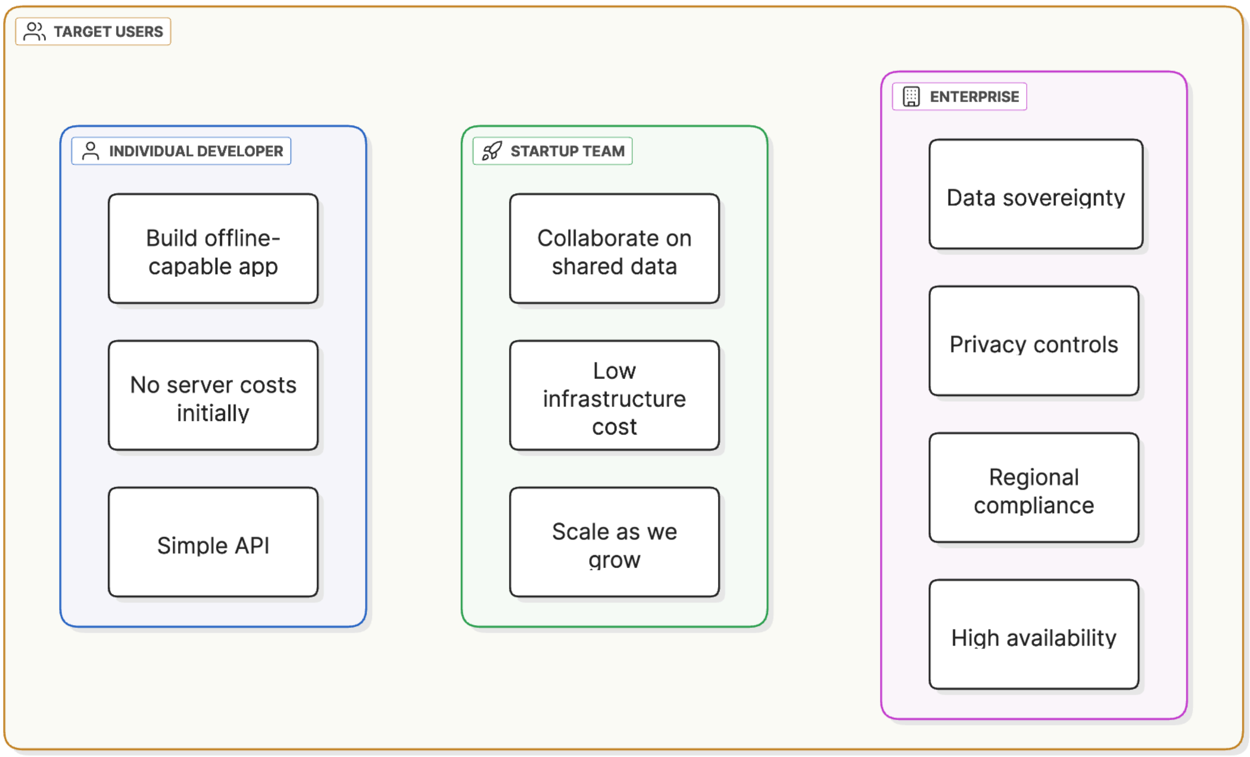
1. **Leverages proven technology**: PouchDB and CouchDB have solved sync for 15+ years

2. **Adds developer experience**: Simple SDK, React hooks, schema validation
3. **Handles conflicts intelligently**: Pluggable strategies including Automerge CRDTs
4. **Scales progressively**: No server for individuals → enterprise clusters for organizations
5. **Respects privacy**: Fine-grained control over what data syncs to which level

---

2. Business Requirements

2.1 Target Users



2.2 Functional Requirements

ID	Requirement	Priority	Rationale
FR-1	Apps must work fully offline	Must Have	Core value proposition
FR-2	Data must sync automatically when online	Must Have	Seamless user experience
FR-3	Conflicts must be resolved automatically	Must Have	Users shouldn't see merge dialogs
FR-4	Individual users need no server	Must Have	Zero barrier to entry
FR-5	Teams can share data via shared server	Must Have	Collaboration use case
FR-6	Privacy levels control sync boundaries	Must Have	Data sovereignty

ID	Requirement	Priority	Rationale
FR-7	SDK provides React hooks	Should Have	Developer experience
FR-8	CRDT support for complex data	Should Have	Rich collaborative editing
FR-9	Schema validation on write	Should Have	Data integrity
FR-10	Support for binary attachments	Could Have	Files, images

2.3 Non-Functional Requirements

ID	Requirement	Target	Rationale
NFR-1	Bundle size (core)	<50KB gzipped	Mobile performance
NFR-2	Time to first interaction	<100ms	Offline-first means instant
NFR-3	Sync latency	<1s for small docs	Real-time feel
NFR-4	Conflict resolution	Automatic, no data loss	User trust
NFR-5	Browser storage quota	Handle ~5GB	Reasonable offline dataset
NFR-6	Concurrent users per server	50+ (Level 1), 500+ (Level 2)	Scale targets

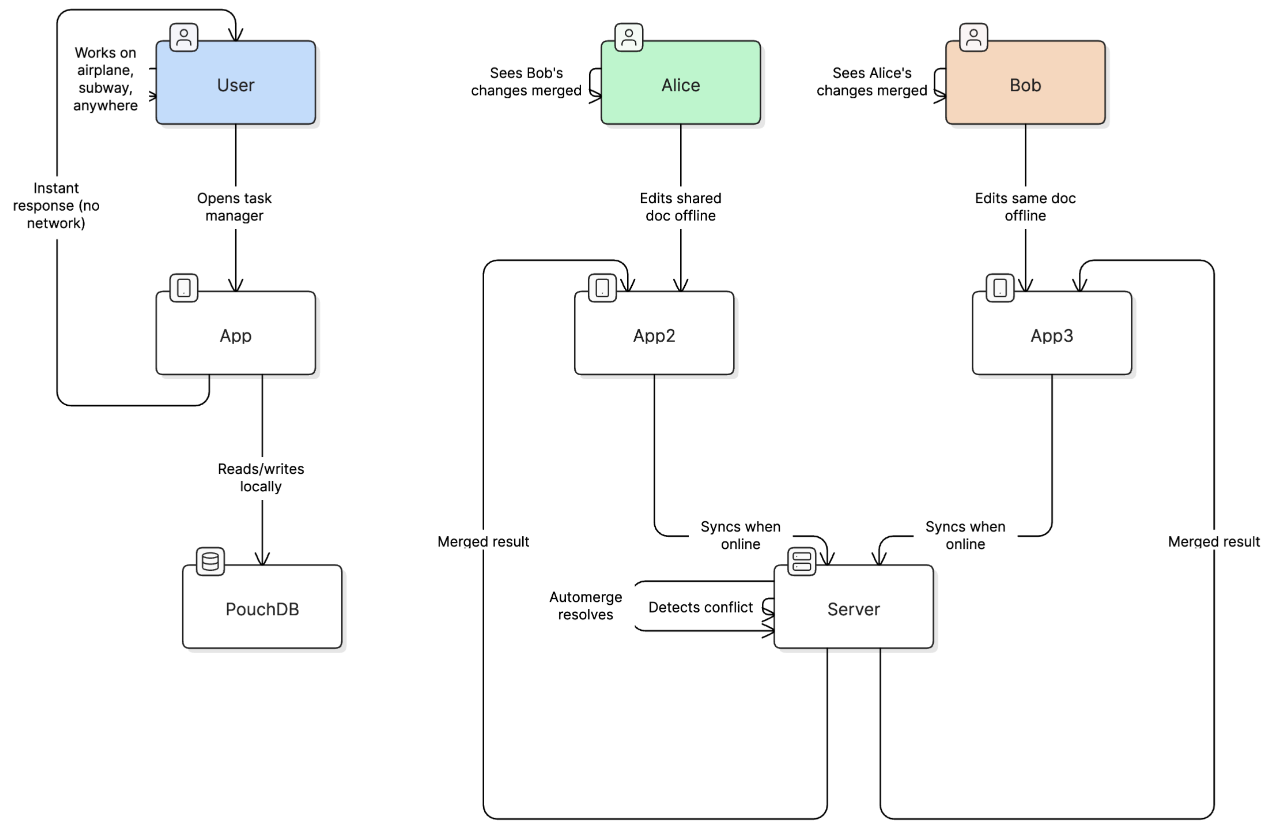
2.4 Business Constraints

Constraint	Impact on Architecture
Minimize custom code	Use PouchDB/CouchDB's built-in sync, don't reinvent
Leverage existing skills	Standard web technologies (TypeScript, React)
Progressive investment	No infrastructure cost until sharing is needed
Proven at scale	CouchDB used by Apple, npm, LinkedIn
Open standards	No vendor lock-in; CouchDB protocol is open

---

3. Use Cases

3.1 Primary Use Cases



3.2 Use Case Matrix

Use Case	Individual	Small Team	Department	Enterprise
Personal notes/tasks	✓			
Family shared lists		✓		
Startup project management		✓		
Field data collection (offline)	✓	✓	✓	
Cross-team collaboration			✓	
Organization-wide directory				✓
Multi-region deployment				✓
Compliance/data residency			✓	✓

3.3 Example Applications

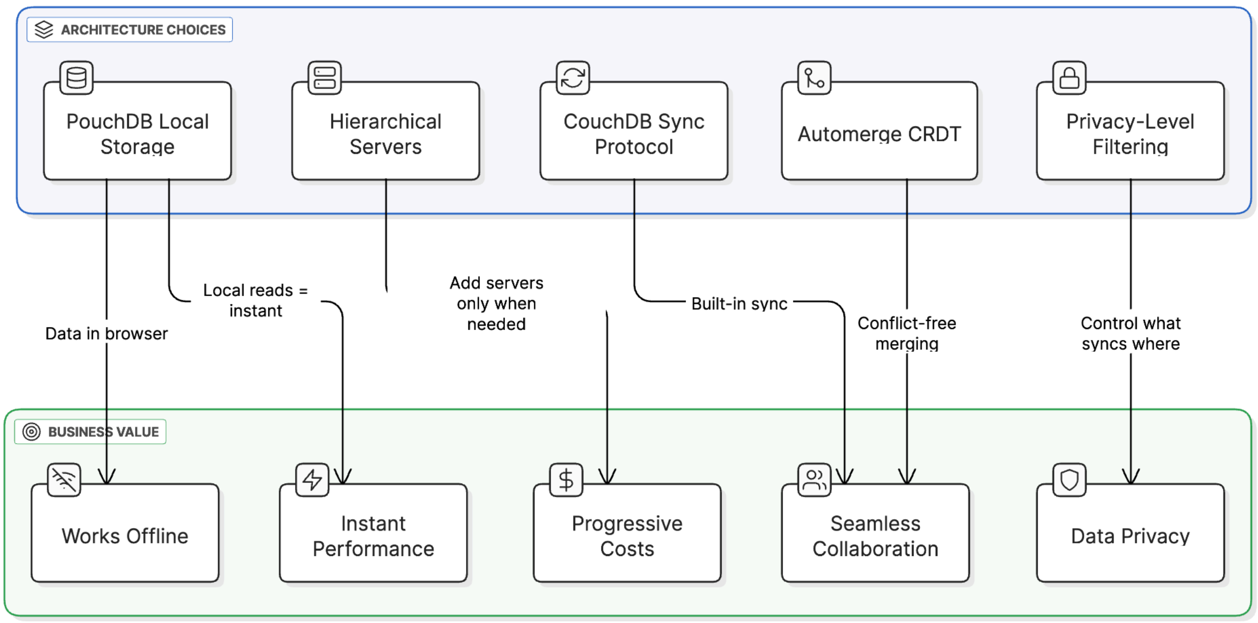
Application Type	Offline Need	Collaboration Need	Recommended Level
Personal journal	High	None	Individual (no server)
Shopping list	Medium	Family sharing	Small Team

Application Type	Offline Need	Collaboration Need	Recommended Level
Inventory tracker	High (warehouse)	Team	Small Team
CRM for sales reps	High (field work)	Department	Department
Document collaboration	Medium	High	Department + CRDT
Enterprise knowledge base	Low	Organization-wide	Enterprise

---

4. Architecture-to-Business Mapping

4.1 How Architecture Delivers Business Value



4.2 Mapping Table

Business Requirement	Architecture Decision	Why This Choice
Apps work offline	PouchDB stores all data locally in IndexedDB	Data available without network
Zero startup cost	Individual level needs no server	PouchDB works standalone
Seamless sync	CouchDB replication protocol (built-in)	Proven, battle-tested, not custom
No merge conflicts for users	Automerge CRDT (optional per collection)	Mathematically guaranteed convergence

Business Requirement	Architecture Decision	Why This Choice
Simple conflict handling	Last-write-wins as default	Zero config for simple apps
Data stays private	Privacy levels on documents	Replication filters enforce boundaries
Scale to enterprise	CouchDB clusters	Same protocol, just more nodes
Small bundle size	Lazy-load Automerge	Only pay for CRDT if you use it
Developer productivity	SDK + React hooks	Familiar patterns, less boilerplate

4.3 Trade-offs Acknowledged

Trade-off	Choice Made	Rationale
Bundle size vs. features	Lazy-load CRDT (~80KB)	Most apps don't need it; load on demand
Simplicity vs. flexibility	Opinionated defaults, escape hatches available	80% of apps use simple patterns
Storage limits	Browser quota (~5GB)	Sufficient for most apps; large data needs different approach
Real-time vs. eventual consistency	Eventual consistency	Simpler, more reliable, works offline

---

5. Architecture Overview

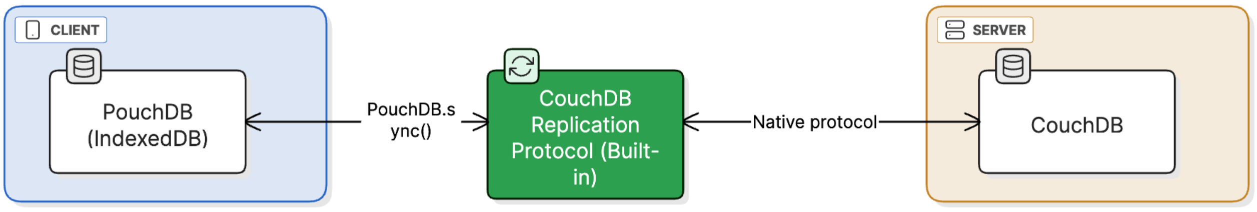
5.1 Design Principles

Principle	Description
Offline-First	Apps work fully offline; sync is opportunistic
Zero to Scale	Individual = no server (PouchDB only); add CouchDB when sharing
Data Sovereignty	Data stays as local as possible; privacy levels control propagation
CouchDB All The Way	Same replication protocol at every level - PouchDB ↔ CouchDB

Principle	Description
Pluggable Conflict Resolution	Automerge CRDT or simpler strategies per collection
Pay-for-What-You-Use	Simple apps stay small; CRDT overhead only when needed

5.2 Core Sync Architecture

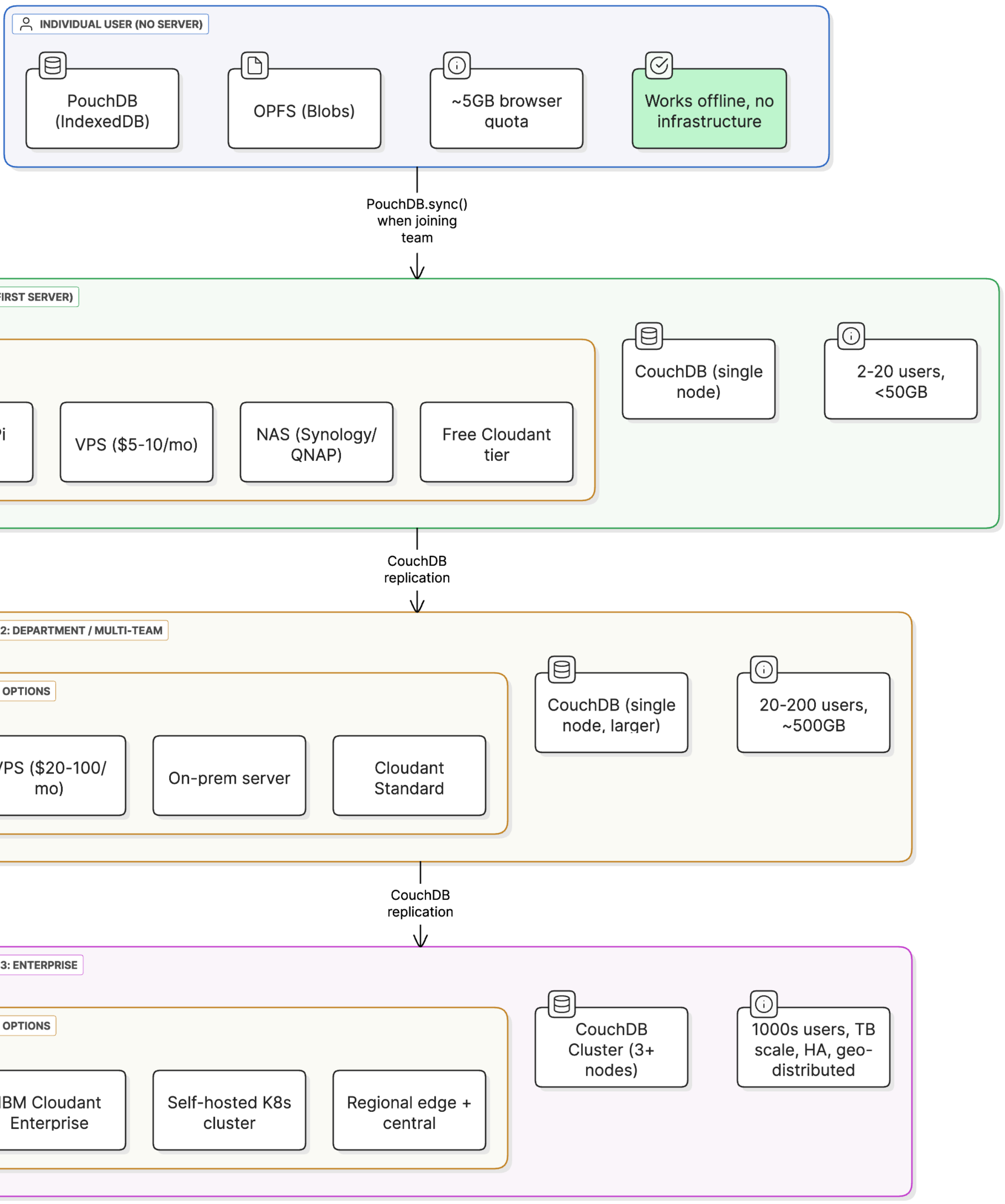
The fundamental architecture is simple: PouchDB syncs to CouchDB using the built-in replication protocol.



This is not a custom sync mechanism. PouchDB and CouchDB speak the same replication protocol natively. Our SDK wraps this with:

- Schema validation
- Privacy-level filtering
- Conflict resolution strategies (including Automerge)
- React bindings

5.3 Storage Strategy: Progressive Infrastructure

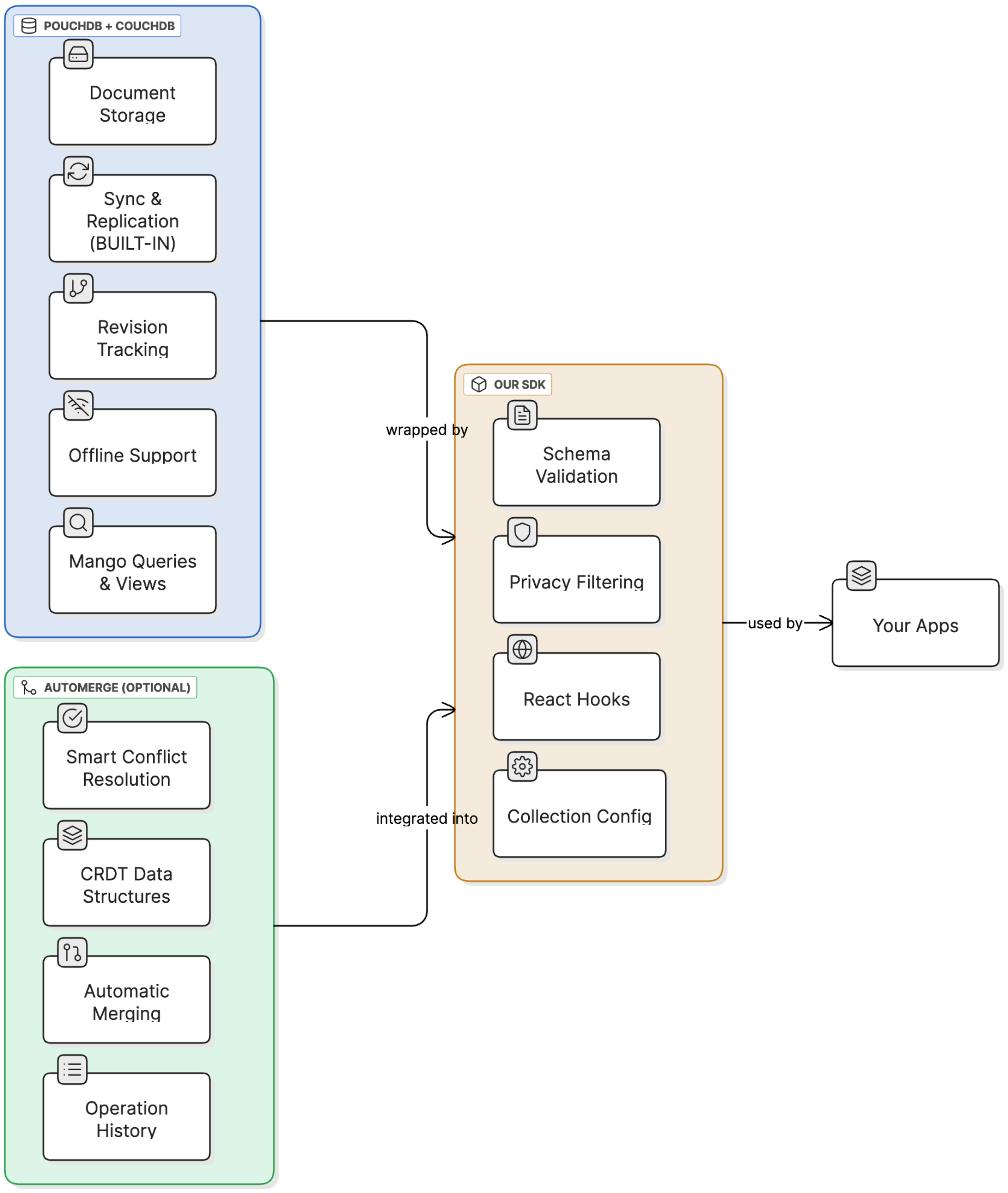


5.4 Storage Hierarchy Summary

Level	Users	Storage	Infrastructure	Use Case
Individual	1	PouchDB (browser)	None	Personal app, single device
Level 1: Small Team	2-20	CouchDB single node	Pi / cheap VPS / NAS	Family, small team, startup
Level 2: Department	20-200	CouchDB single node (beefier)	Dedicated server	Department, multi-team collaboration
Level 3: Enterprise	1000s	CouchDB Cluster	HA infrastructure	Organization-wide, geo-distributed

Key insight: You only need a server when you need to share. Individual users work entirely offline with PouchDB.

5.5 Technology Roles: What Does What?



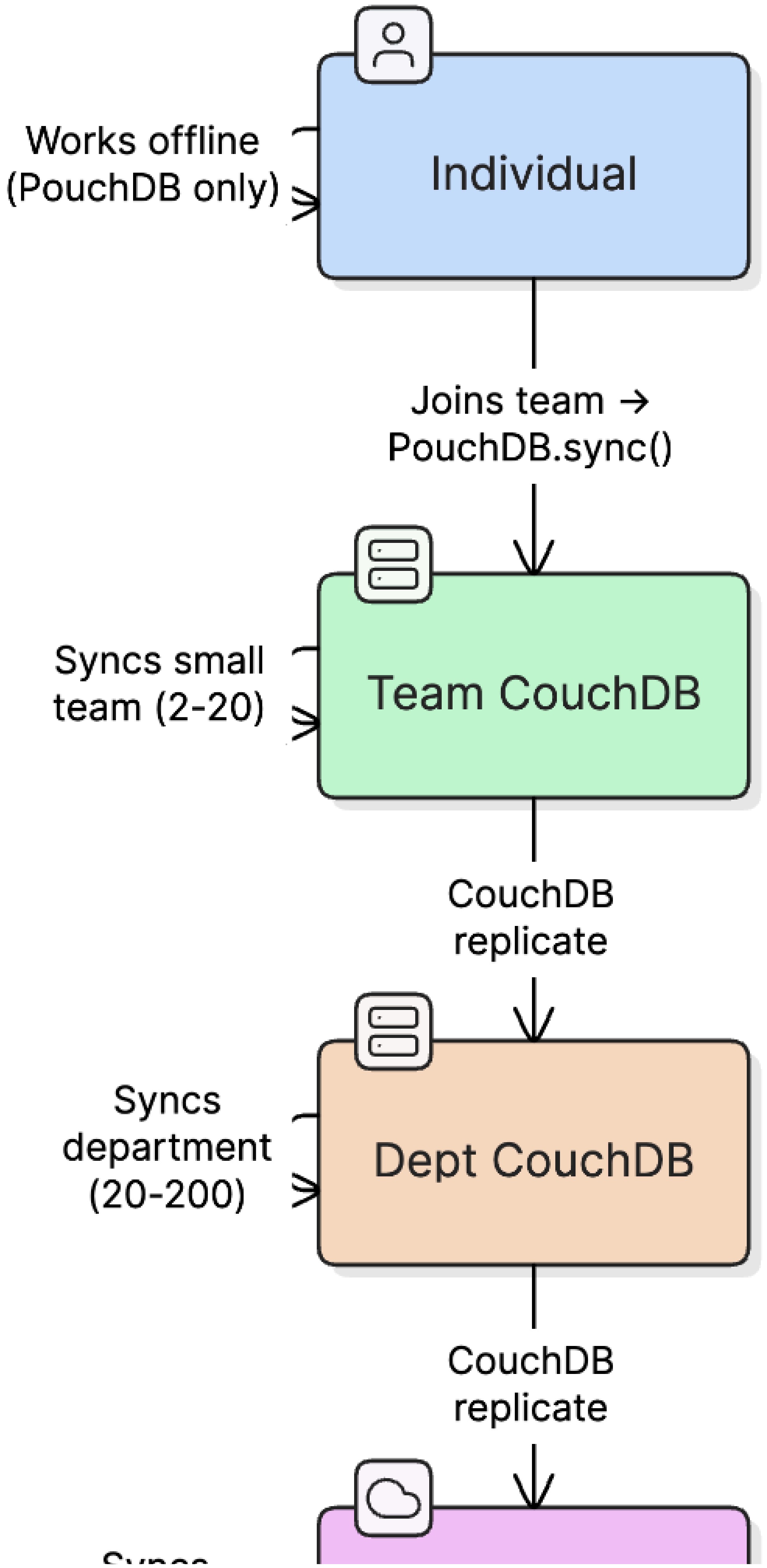
Key insight: We are NOT building a sync engine. We are building a **developer experience layer** on top of PouchDB/CouchDB's existing sync.

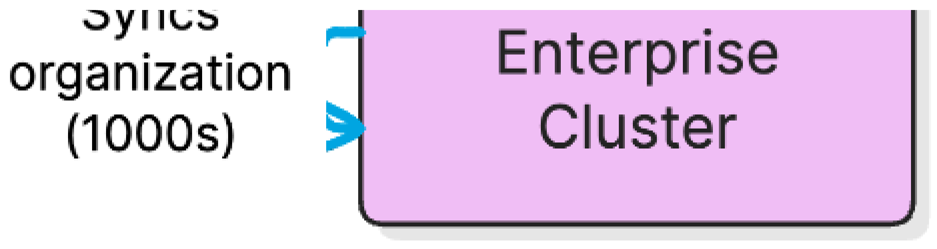
---

6. Hierarchical Sync Architecture

6.1 How Hierarchical Sync Works

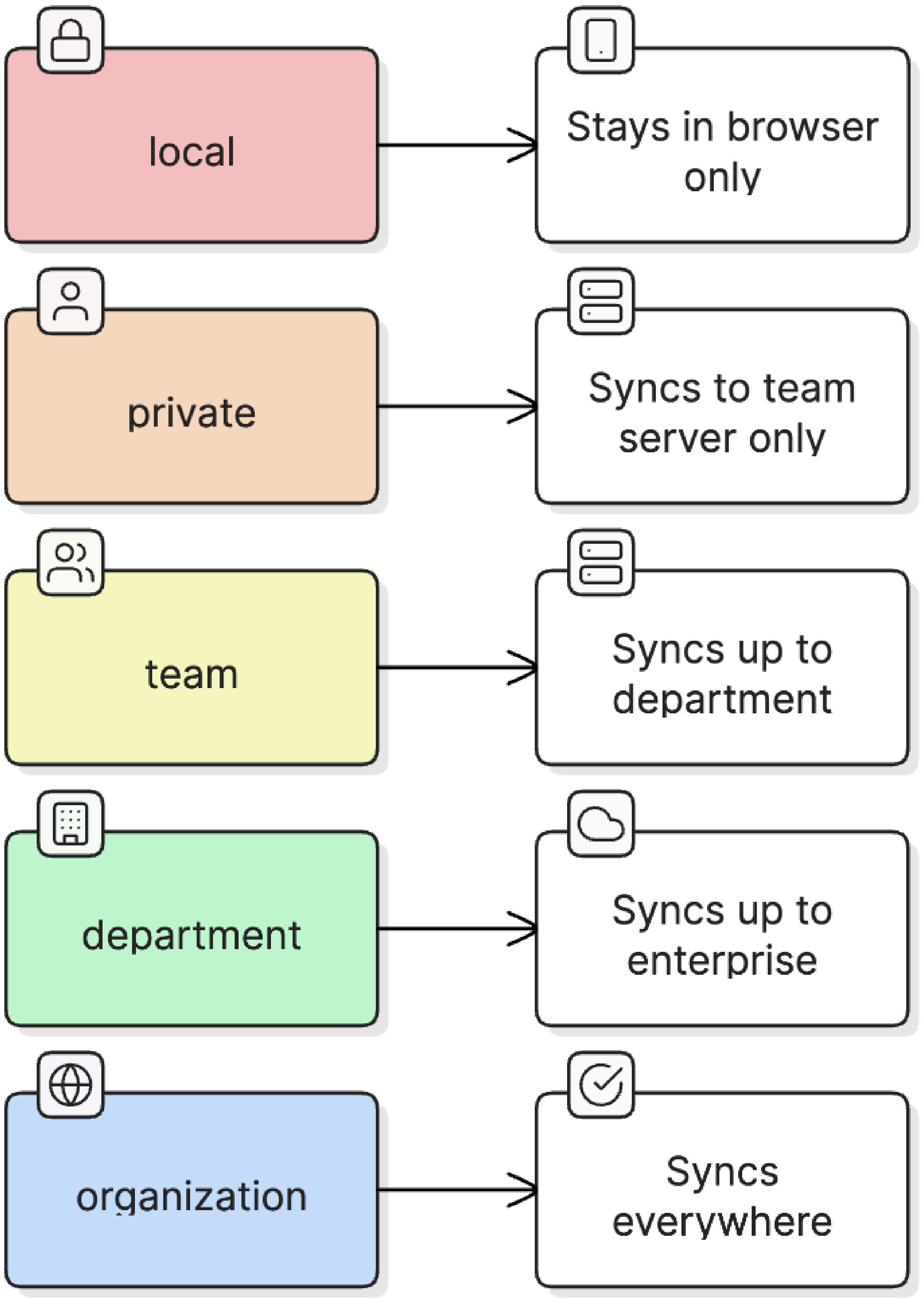
Individual users work locally. When they join a team, they sync to that team's CouchDB. Teams can sync up to departments, and departments to enterprise:


Works offline (PouchDB only)
Individual
Joins team → PouchDB.sync()
Syncs small team (2-20)
Team CouchDB
CouchDB replicate
Syncs department (20-200)
Dept CouchDB
CouchDB replicate



6.2 Privacy-Based Filtering

Documents have a `privacyLevel` that controls how far up they sync:

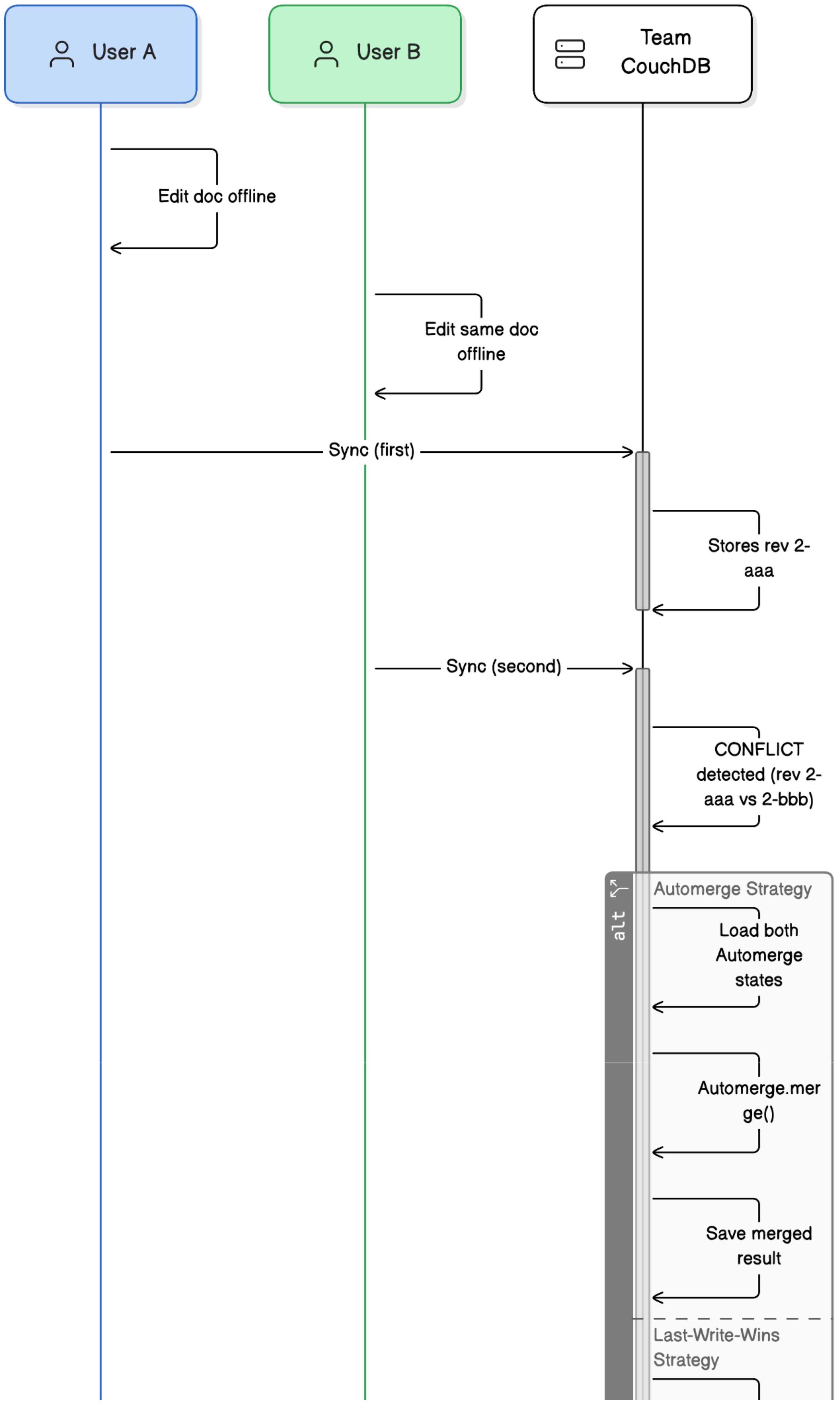


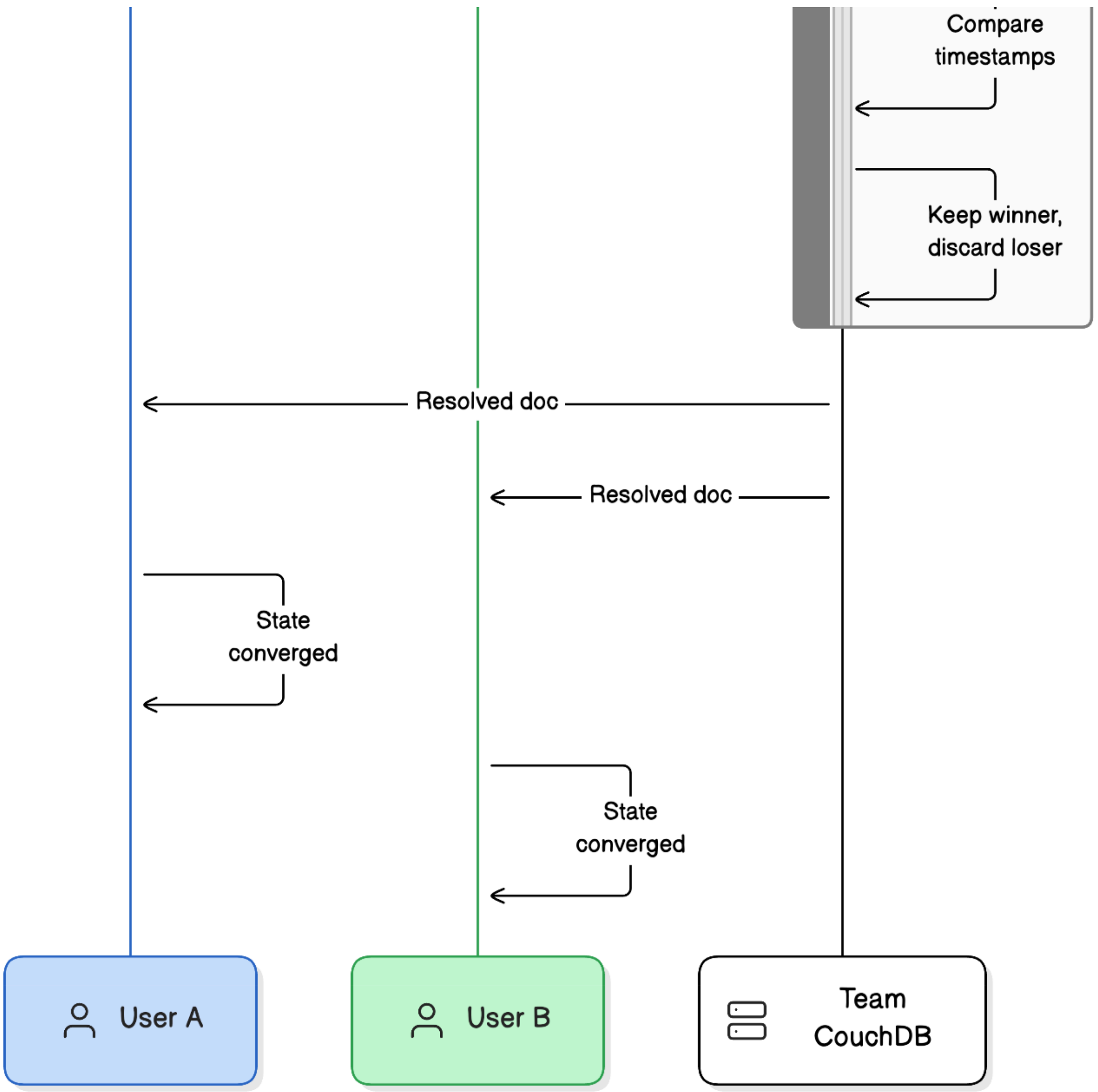
Replication filters at each CouchDB instance enforce these boundaries:

```
// Filter function installed on Department CouchDB
function(doc, req) {
  // Only replicate docs with department+ privacy to enterprise
  var validLevels = ['department', 'organization'];
  return validLevels.indexOf(doc._syncMeta.privacyLevel) !== -1;
}
```

6.3 Conflict Resolution in the Hierarchy

When conflicts occur (simultaneous offline edits), they're detected by CouchDB's revision system and resolved by our configured strategy:


User A
User B
Team CouchDB
Edit doc offline
Edit same doc offline
Sync (first)
Stores rev 2-aaa
Sync (second)
CONFLICT detected (rev 2-aaa vs 2-bbb)
alt
Automerge Strategy
Load both Automerge states
Automerge.merge()
Save merged result
Last-Write-Wins Strategy



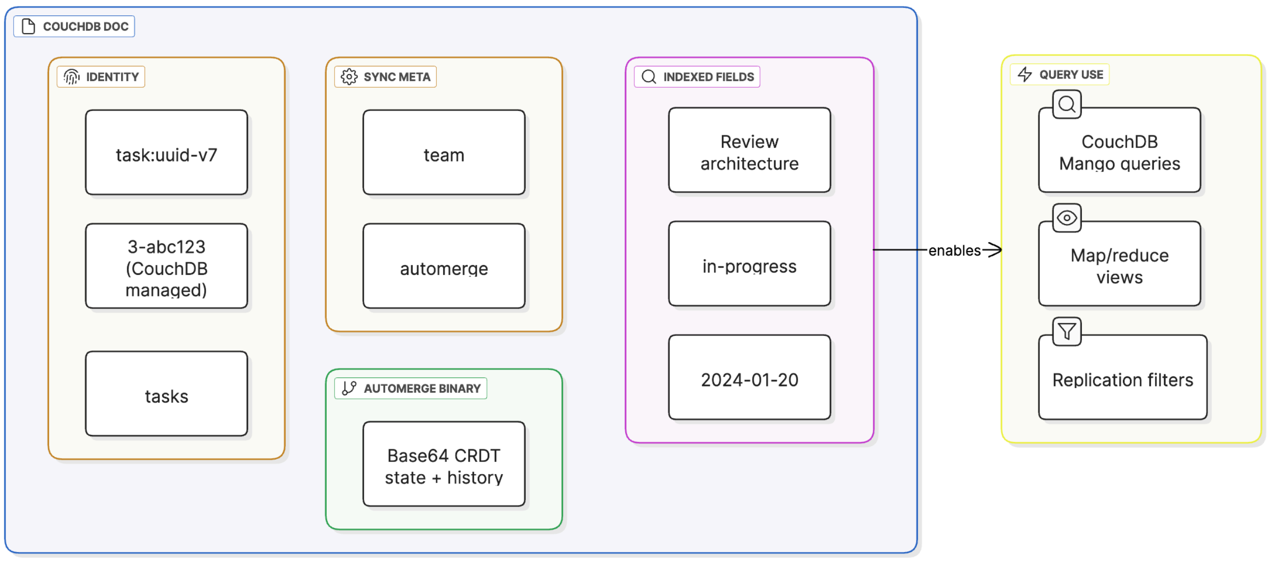
7. Data Model Architecture

7.1 Document Structure

Documents support two modes: **Standard** (plain JSON) and **CRDT** (Automerge-wrapped):

```
// Base document interface
interface BaseDocument {
  _id: string;                       // Unique identifier
  _rev: string;                      // CouchDB revision (managed by
PouchDB/CouchDB)
  _namespace: string;                // Namespace for multi-app support
  _collection: string;               // Collection name

  _syncMeta: {
    origin: string;                  // Node ID that created this revision
    timestamp: number;               // HLC timestamp
    privacyLevel: PrivacyLevel;      // Controls sync propagation
    conflictStrategy: ConflictStrategy;
  };
}

// Standard document - plain JSON
interface StandardDocument extends BaseDocument {
  data: Record<string, any>;
}

// CRDT document - Automerge-wrapped
interface CRDTDocument extends BaseDocument {
  _automerge: string;              // Base64 Automerge binary
  _indexed: Record<string, any>;   // Extracted fields for queries
}

type PrivacyLevel = 'local' | 'private' | 'team' | 'department' |
'organization';

type ConflictStrategy =
  | 'last-write-wins'    // Simple, may lose data
  | 'field-merge'        // Merge non-conflicting fields
  | 'keep-both'          // Manual resolution
  | 'automerge'          // CRDT auto-merge
  | 'automerge-text'     // CRDT with text support
  | 'custom';            // App-provided resolver
```

7.2 CRDT Document Structure

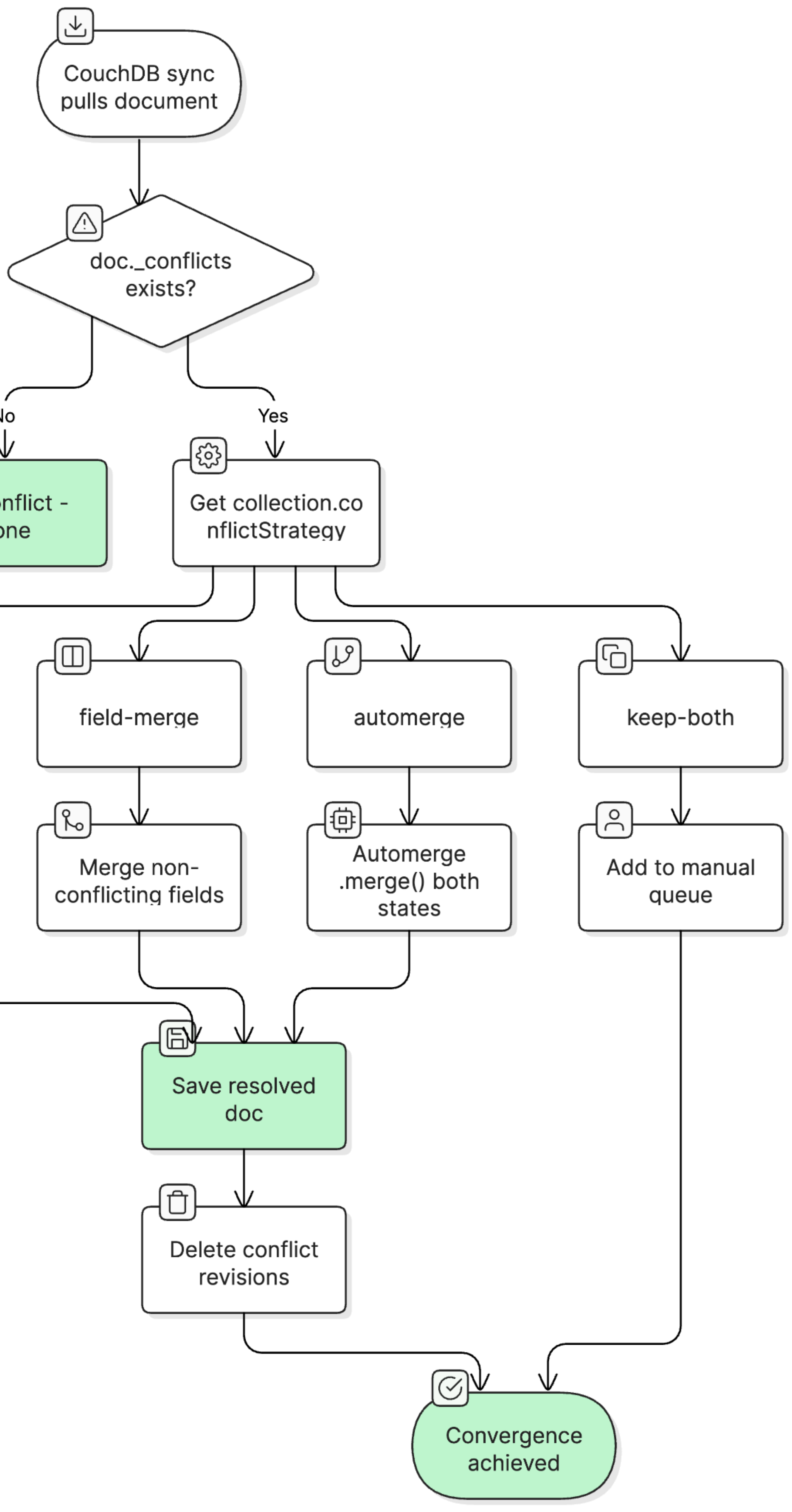


8. Conflict Resolution Strategies

8.1 Strategy Selection Per Collection

Strategy	Use Case	Data Loss Risk	Bundle Cost
last-write-wins	Settings, preferences	High	None
field-merge	Forms, profiles	Medium	~2KB
keep-both	Legal docs, critical data	None (manual)	None
automerge	Collaborative data	None (auto)	~200KB WASM
automerge-text	Rich text editing	None (auto)	~200KB WASM

8.2 Conflict Resolution Flow


CouchDB sync pulls document
doc._conflicts exists?
No
Yes
No conflict - done
Get collection.conflictStrategy
last-write-wins
field-merge
automerge
keep-both
Compare timestamps, keep newest
Merge non-conflicting fields
Automerge .merge() both states
Add to manual queue
Save resolved doc
Delete conflict revisions
Convergence achieved

8.3 Automerge Resolution Code

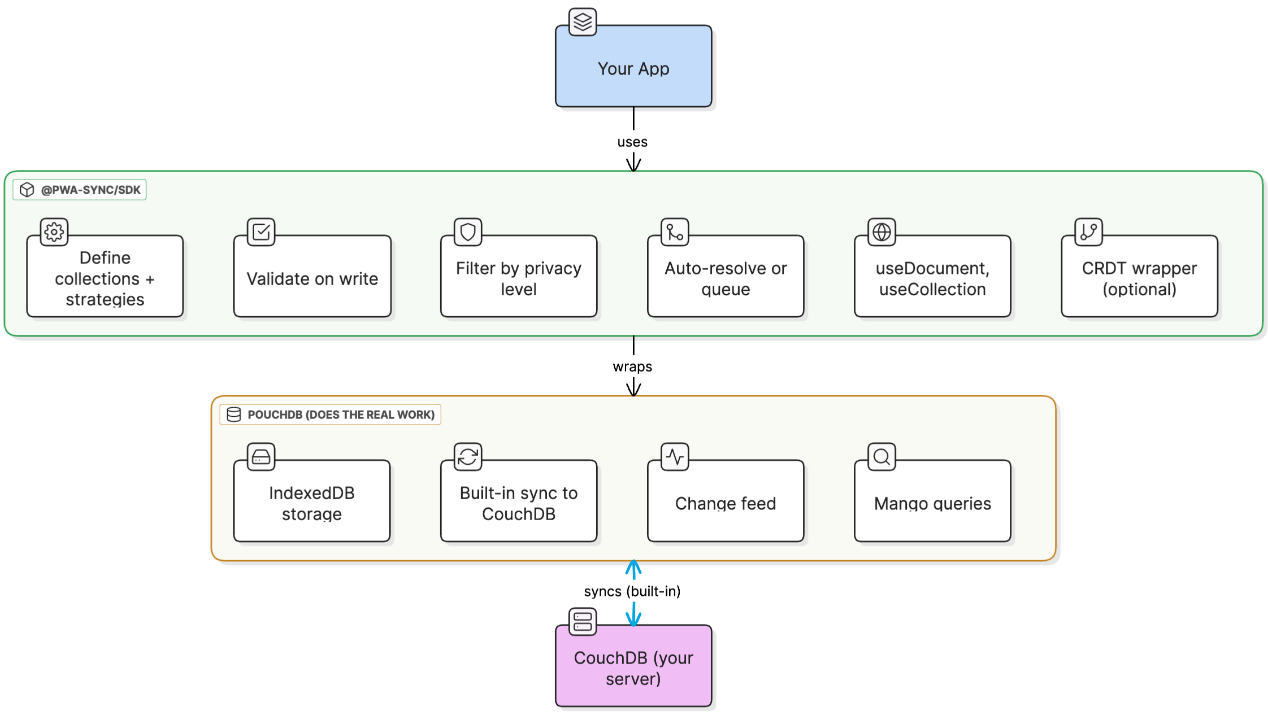
```
import * as Automerge from '@automerge/automerge';

async function automergeResolve(
  db: PouchDB.Database,
  doc: CRDTDocument,
  conflicts: string[]
): Promise<void> {
  // Load winner's state
  let merged = Automerge.load(base64ToBytes(doc._automerge));

  // Merge each conflicting revision
  for (const rev of conflicts) {
    const conflictDoc = await db.get(doc._id, { rev });
    const conflictState =
Automerge.load(base64ToBytes(conflictDoc._automerge));
    merged = Automerge.merge(merged, conflictState);
  }

  // Save merged result
  await db.put({
    ...doc,
    _automerge: bytesToBase64(Automerge.save(merged)),
    _indexed: extractIndexedFields(merged)
  });

  // Delete conflict revisions
  for (const rev of conflicts) {
    await db.remove(doc._id, rev);
  }
}
```

---

9. SDK Architecture

9.1 What the SDK Provides

The SDK is a **thin wrapper** around PouchDB that adds:



9.2 SDK API

```
import { createSyncSDK } from '@pwa-sync/sdk';

const sdk = createSyncSDK({
  dbName: 'my-app',

  // Optional: configure remote CouchDB for sync
  remote: {
    url: 'https://my-couchdb.example.com/mydb',
    auth: { type: 'jwt', tokenProvider: getToken }
  },

  // Define collections with conflict strategies
  collections: {
    settings: {
      conflictStrategy: 'last-write-wins'  // Simple, no CRDT
    },
    tasks: {
      conflictStrategy: 'automerge',         // Smart merge
      indexedFields: ['title', 'status', 'dueDate']
    },
    documents: {
      conflictStrategy: 'automerge-text'     // Rich text
    }
  }
});

// CRUD operations
const task = await sdk.create('tasks', { title: 'Review PR', status:
```

```
'pending' });
await sdk.update('tasks', task._id, doc => { doc.status = 'done'; });
await sdk.delete('tasks', task._id);

// Queries (uses indexed fields for CRDT collections)
const pending = await sdk.query('tasks', {
  filter: { status: 'pending' }
});

// Sync control
await sdk.sync();                    // Manual sync
sdk.enableAutoSync({ interval: 30000 }); // Auto sync

// Sync is just PouchDB.sync() under the hood!
```

9.3 React Hooks

```
import { SyncProvider, useDocument, useCollection, useSyncStatus } from
'@pwa-sync/sdk/react';

function App() {
  return (
    <SyncProvider sdk={sdk}>
      <TaskList />
    </SyncProvider>
  );
}

function TaskList() {
  const { data: tasks, loading } = useCollection('tasks', {
    filter: { status: 'pending' }
  });

  const { status, pendingChanges } = useSyncStatus();

  return (
    <div>
      <SyncIndicator status={status} pending={pendingChanges} />
      {tasks.map(task => <TaskItem key={task._id} id={task._id} />)}
    </div>
  );
}

function TaskItem({ id }) {
  const { data, update } = useDocument('tasks', id);

  return (
    <div onClick={() => update(d => { d.status = 'done'; })}>
      {data.title}
    </div>
  );
}
```

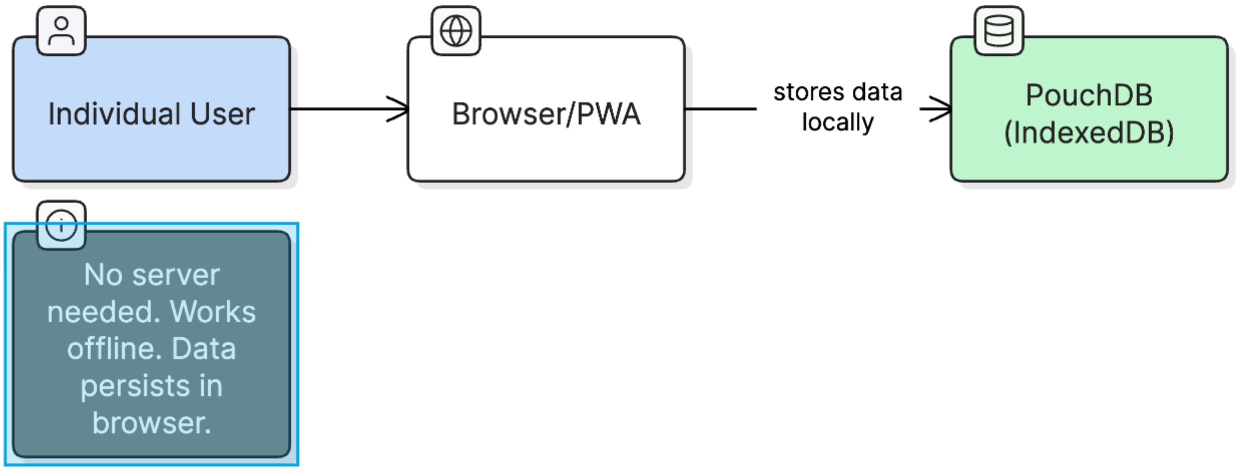
9.4 Bundle Size

Component	Size (gzipped)	When Loaded
Core SDK	~15KB	Always
PouchDB	~35KB	Always
React bindings	~5KB	If using React
Automerge WASM	~80KB	On first CRDT operation (lazy)
Simple app	**~50KB**	No CRDT collections
Full app	**~135KB**	With CRDT collections

---

10. Deployment Scenarios

10.1 Scenario: Individual User (No Server)

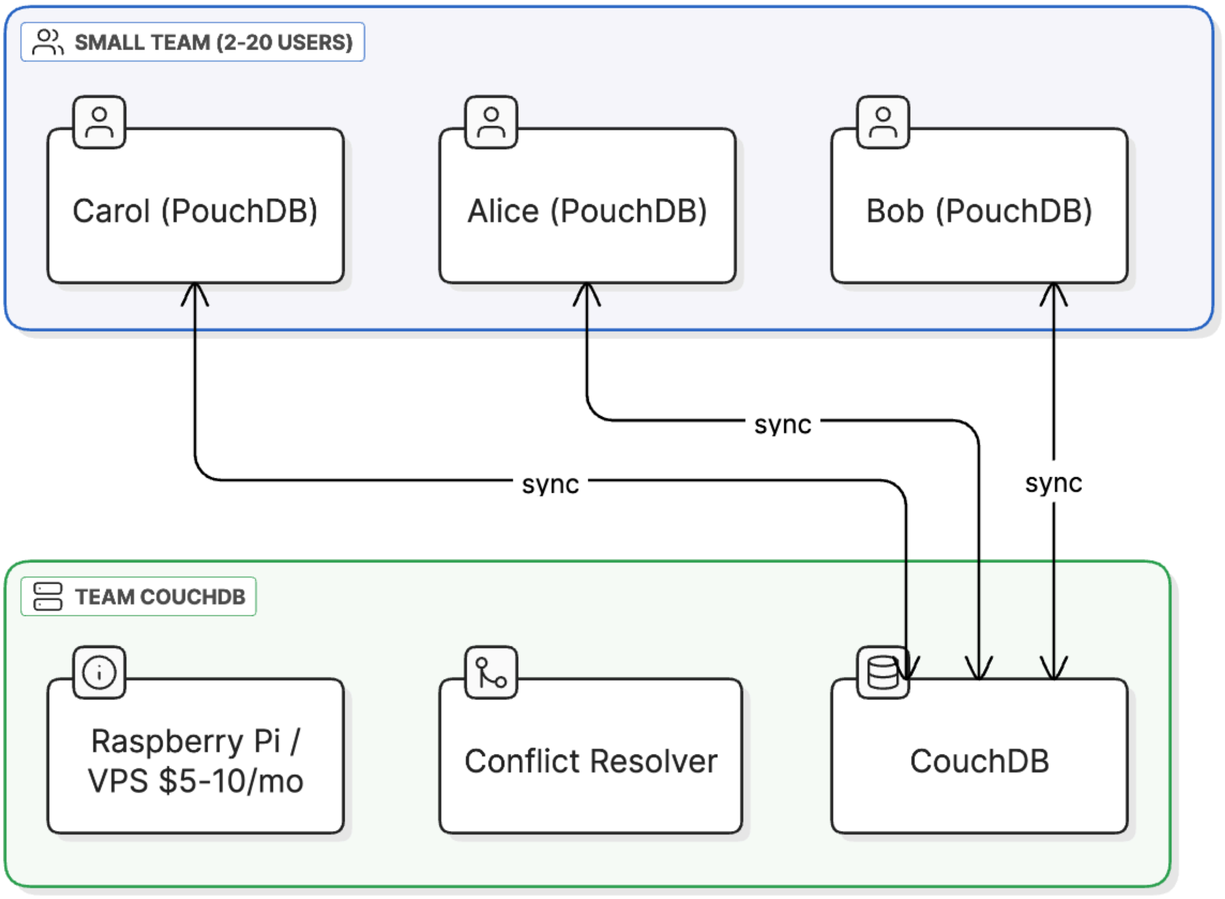


Use case: Personal notes, single-user tools, prototyping, offline-first apps.

Code:

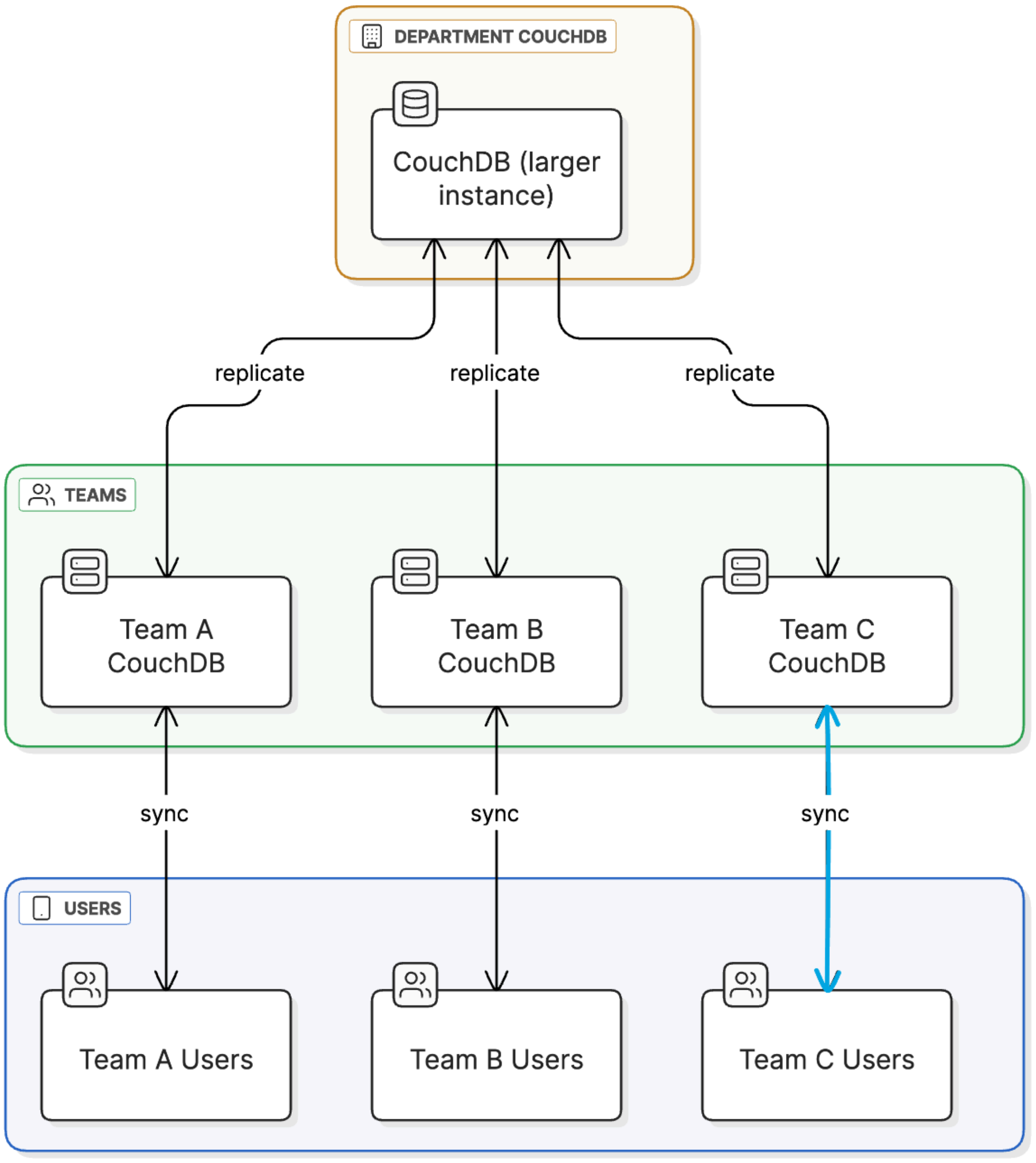
```
const sdk = createSyncSDK({
  dbName: 'my-app',
  // No 'remote' = purely local
  collections: { tasks: { conflictStrategy: 'last-write-wins' } }
});
```

10.2 Scenario: Small Team (Level 1)



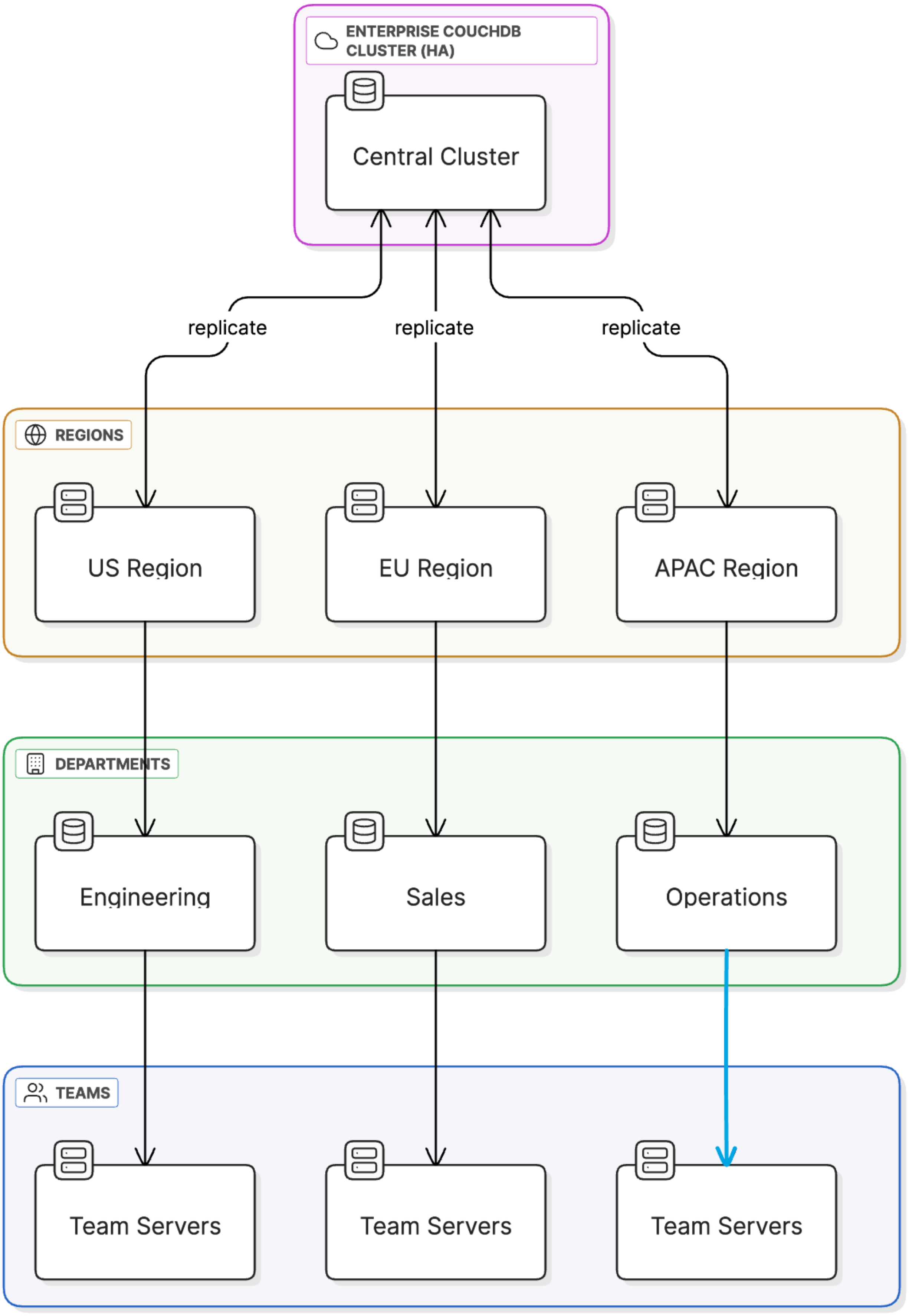
Use case: Family shared lists, startup team, small project collaboration.

10.3 Scenario: Department / Multi-Team (Level 2)



Use case: Multiple teams collaborating, department-wide tools, cross-team projects.

10.4 Scenario: Enterprise (Level 3)



Use case: Global enterprise, regional data residency, thousands of users, high availability.

11. Server Setup

11.1 Docker Compose for Team Server

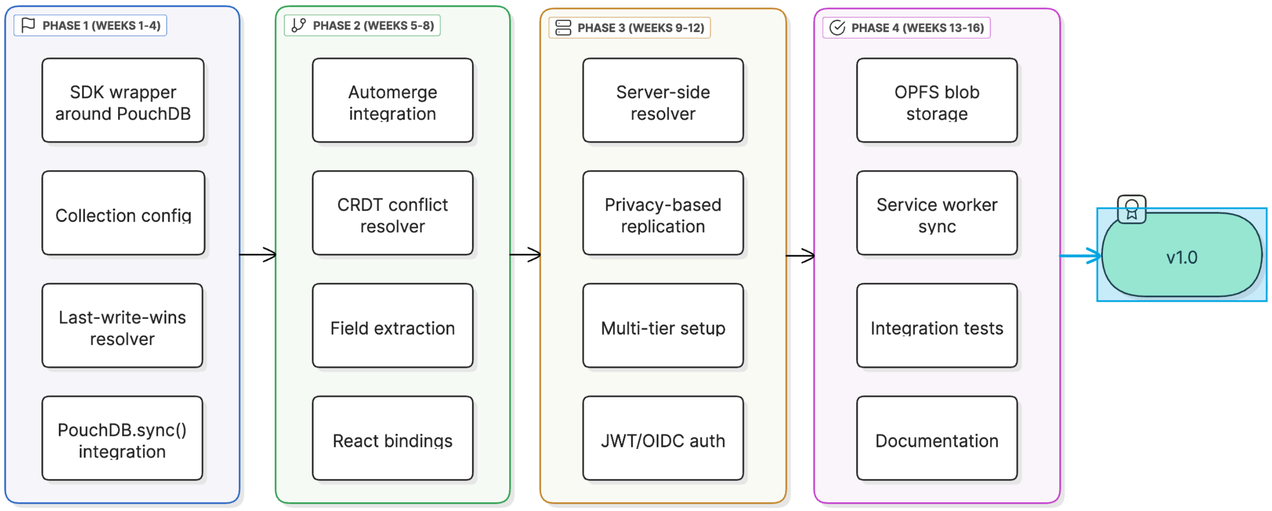
```yaml
version: '3.8'

services:
  couchdb:
    image: couchdb:3.3
    restart: unless-stopped
    environment:
      - COUCHDB_USER=${COUCHDB_USER:-admin}
      - COUCHDB_PASSWORD=${COUCHDB_PASSWORD}
    volumes:
      - couchdb_data:/opt/couchdb/data
    ports:
      - "5984:5984"

  # Optional: server-side conflict resolver for CRDT collections
  conflict-resolver:
    image: pwa-sync/conflict-resolver:latest
    restart: unless-stopped
    depends_on:
      - couchdb
    environment:
      - COUCHDB_URL=http://couchdb:5984
      - COUCHDB_USER=${COUCHDB_USER}
      - COUCHDB_PASSWORD=${COUCHDB_PASSWORD}
      - CRDT_COLLECTIONS=tasks,documents

volumes:
  couchdb_data:
```

11.2 Raspberry Pi Setup

```bash
# Install Docker
curl -fsSL https://get.docker.com | sh
sudo usermod -aG docker pi

# Run CouchDB
docker run -d \
  --name couchdb \
  --restart unless-stopped \
  -p 5984:5984 \
  -e COUCHDB_USER=admin \
  -e COUCHDB_PASSWORD=your-password \
  -v couchdb_data:/opt/couchdb/data \
  couchdb:3.3
```

```
# Access at http://raspberrypi.local:5984
```

12. Implementation Roadmap



13. Key Decisions

Decision	Choice	Rationale
Sync Mechanism	PouchDB's built-in sync	Battle-tested, no need to reinvent
Server Storage	CouchDB at all levels	Same protocol, simple architecture
CRDT Library	Automerge (optional)	Best JSON CRDT, good merge semantics
CRDT Integration	Per-collection opt-in	Pay-for-what-you-use, smaller bundles
Conflict Detection	CouchDB revisions	Native support, works automatically

14. Summary

The architecture is intentionally simple:

1. **Individual**: PouchDB stores data locally in browser - no server needed
2. **Small Team (Level 1)**: Add a CouchDB instance, users sync via PouchDB.sync()
3. **Department (Level 2)**: Teams replicate to department CouchDB
4. **Enterprise (Level 3)**: Departments replicate to enterprise CouchDB cluster
5. **Conflicts**: CouchDB detects, our resolvers handle (Automerge or simpler)

6. **SDK**: Thin wrapper adding schema, privacy, React hooks

Storage Hierarchy:

Level	Users	Server	Sync Mechanism
Individual	1	None (PouchDB only)	N/A
Small Team	2-20	CouchDB single node	PouchDB.sync()
Department	20-200	CouchDB single node	CouchDB replication
Enterprise	1000s	CouchDB Cluster	CouchDB replication

We are NOT building a sync engine. We are building developer tools on top of the proven PouchDB ↔ CouchDB sync.